CS 422 (Park) **Assignment IV** **Due: Apr. 6 (Tue.), 2004**

Additional submission instructions: For code implementations, organize your code around a Makefile that automates compilation. All source and object code should be put in subdirectories. Use subdirectories to organize your output files. The current directory should be empty except for Makefile, a README file, and subdirectories. The README file should briefly describe what is contained in the subdirectories. Makefile should create the executable(s) in the current directory. Your code will be tested by automated scripts that check compilation and run-time behavior. Your source code will be tested manually and by automated scripts. Consult the TA's link for any additional instructions.

PROBLEM 1 (15 pts)

Read Chapters 11, 18, 20, 21, 24, 25, 29 and 30 from Comer. Solve Problems 11.12, 21.4, 24.1, and 25.4.

PROBLEM 2 (40 pts)

As a continuation of Problem 3(a), Assignment III, rewrite the application so that it uses TCP instead of FIFOs to communicate between server and (multiple) clients across an IP internetwork. The server, when starting up, should take a command-line argument indicating the port number at which it listens to client requests. The request format should change to

 # IP-address # port-number # command # argument-1 *# argument*-2 *# ⋯ # argument-n #*

where *IP-address* and *port-number* denote the IP address and port number of the client, respectively. `client.bin`'s first two command-line arguments should comprise of the server's IP address (in dotted decimal notation) and port number. The remaining arguments represent the command to be executed by the server as before. The server, after parsing a client request but before handing it off to a child process, should print onto the terminal its own IP address (in dotted decimal form), port number, the client's IP address and port number, the command to be executed, and a time stamp of when the client request was received (in human legible format up to millisecond granularity).

Test your program by first running the server application on a `xinu` machine, then executing the client `client.bin` on separate machines with `date`, `host`, `ps -l`, `ps -l -a`, `ls`, `ls -a -l -i`, and `servus`. Start the 7 client programs concurrently. Thus a total of 8 machines are involved (1 server and 7 clients). Use `script` to record the run-time session at the server and 7 clients. Submit your code and output (see instructions above). You must provide adequate documentation in your code.

PROBLEM 3 (50 pts)

Write a UDP-based CBR (constant bit rate) traffic generation application where the sender, `cbr_sender`, transmits packets of a fixed size at fixed rate while the receiver, `cbr_receiver`, receives the packets. The sender creates a log file with three columns where the first column records the sequence number of the packet just sent (1, 2, ...), the second column the current time stamp, and the third column the size of the packet (UDP's payload only). The receiver, upon processing an arriving packet, also records into a log file of the same format. The sequence number of the packet is obtained from the first 4 bytes of the payload in which the sender inscribes the sequence number.

`cbr_sender` takes the command-line arguments *IP-address*, *port-number*, *packet-spacing*, *packet-size*, *packet-count*, and *log-file-name*. *IP-address* and *port-number* specify the receiver's coordinates, *packet-spacing* (μsec granularity) denotes the time interval between successive packets, *packet-size* (in bytes) denotes the size of the UDP payload (at least 4 bytes), *packet-count* specifies how many packets are to be sent, and *log-file-name* denotes the name of the log file. The sender's basic structure is a `usleep` loop which goes off every *packet-spacing* microseconds at which a packet is sent off and the event recorded. The receiver's basic structure is an infinite wait that blocks until SIGPOLL (equivalently SIGIO) is raised at which time a signal handler fetches the newly arrived packet and records the event in a log file. Before returning, the signal handler checks if any other packets have arrived while it was processing the packet and, if so, processes the remaining queued packets until no more packets remain. The receiver takes the port number on which it should wait as a command-line argument. Both sender and receiver should maintain the logs in memory and only flush to disk at the end of the run to reduce slow-down due to disk I/O.

Benchmark your application using two `xinu` machines for sender and receiver, respectively. The first test suite should use *packet-size* 1 KB, with *packet-count* and *packet-spacing* combos 300 and 200000 μsec, 400 and 150000 μsec, 600 and 100000 μsec, 1200 and 50000 μsec, and 60000 and 1000 μsec. For each combo, use `gnuplot` to create a plot where the x-axis represents time at 1 sec granularity (you need to preprocess the log file so that events are aggregated in 1 second blocks) and the y axis represents data rate (bps). In each plot, draw three graphs: one, showing the sender's recorded data rate, two showing the receiver's recorded data rate, and three, showing the ideal data rate (calculated by multiplying packets-per-second with packet size in bits). Include the UDP header and Ethernet header/trailer in your data rate calculations for all three graphs. Along with the plots, give a 1/2–1 page interpretation of the results. Repeat the benchmark runs with *packet-size* 4 KB and 512 B. How do the plots compare as the packet size is varied?

EXTRA CREDIT PROBLEM (20 pts)

What is the highest data rate that you can achieve with the CBR traffic generator, and at what values of *packet-size* and *packet-spacing* is it achieved? Show the plots. The aim is to pick a combo that has as small a packet size and as large a packet spacing as possible while maximizing the achieved throughput. Using your knowledge of operating systems, provide explanations for the "discrepancies" you observe between the ideal rate and the observed sender/receiver rates, and why beyond a certain *packet-spacing* threshold no further increase in data rate is attainable. What kernel parameter can be reconfigured to achieve a packet spacing of 1 msec?